Projektowanie Efektywnych Algorytmów

Kierunek *Informatyka*	**Termin** *Czwartek 19:05*
Temat *Algorytmy dokładne*	**Problem** *TSP*
Skład grupy *241281 Karol Kulawiec*	**Nr grupy** -
Prowadzący *Mgr inż. Radosław Idzikowski*	**data** *9 kwietnia 2020*

1 Opis problemu

Podczas pierwszego etapu należało napisać algorytmy dokładne rozwiązujące problem komiwojażera. Problem komiwojażera polega na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Jest to problem NP-trudny, co oznacza, że nie jest możliwe rozwiązanie problemu komiwojażera ze złożonością obliczeniową wielomianową.
Zbadane algorytmy dokładne: Brute Force, Dynamic Programming (Held-Karp) oraz Branch and Bound.

2 Metoda rozwiązania

2.1 Brute Force

Algorytm Brute Force, to najprostszy algorytm. Jego działanie polega na przeglądnięciu wszystkich możliwych ścieżek i wybraniu tej najkrótszej. Aby algorytm wiedział, długość której ścieżki ma zmierzyć, należy te ścieżki albo wyznaczyć wcześniej, albo wyznaczać po skończeniu operacji z bieżącą ścieżką. W programie wykorzystano drugi sposób, ponieważ pozwala on zmniejszyć wykorzystanie pamięci. W języku Python jest to możliwe dzięki generatorom, które tworzą kolejne elementy dopiero przy odwołaniu się do nich (np. poprzez pobranie kolejnego elementu).

Do wytworzenia kolejnych permutacji ścieżek wykorzystano algorytm Heap Permutation, który w wersji z generatorem wygląda następująco:

Listing 1: Algorytm Heap Permutation

```
1 def heap_perm(n, A):
2     if n == 1: yield A
3     else:
4         for i in range(n-1):
5             for hp in heap_perm(n-1, A): yield hp
6             j = 0 if (n % 2) == 1 else i
7             A[j],A[n-1] = A[n-1],A[j]
8         for hp in heap_perm(n-1, A): yield hp
```

gdzie *A* to lista wierzchołków, a *n* to rozmiar tej listy.

Do zliczenia długości ścieżki użyto następującej funkcji:

Listing 2: Algorytm zliczający długość ścieżki

```
1 def countPath(path):
2     new_path=0
3     new_path+=arr[0][path[0]]
4     for step in range(n-2):
5         new_path+=arr[path[step]][path[step+1]]
6     new_path+=arr[path[n-2]][0]
7     return new_path
```

gdzie *path* to lista sąsiadujących wierzchołków, a *arr* to macierz, w której są zapisane odległości pomiędzy poszczególnymi wierzchołkami.

2.1.1 Algorytm i jego przebieg

W algorytmie najpierw zadeklarowano zmienną *b* jako generatora (1), następnie pobrano pierwszą z permutacji i obliczono jej długość (2-4), tak aby w kolejnych krokach móc porównać nowe wartości. W pętli, która wyko-

nuje się $((n-1)!-1)$ razy (6) za każdym razem pobieramy nową ścieżkę (7), obliczamy jej długość (8) oraz sprawdzamy, czy jest krótsza od dotychczas najkrótszej (9-11).

Listing 3: Algorytm Brute Force

```
1  b=heap_perm(n-1,list(range(1,n)))
2  path = next(b)
3  min_path = path
4  min_length = countPath(path)
5
6  for i in range(math.factorial(n-1)-1):
7      path = next(b)
8      length = countPath(path)
9      if length < min_length:
10         min_length = length
11         min_path = path
```

Złożoność algorytmu Brute Force to $O(n!)$, co sprawia, że już dla małych instancji, algorytm długo się wykonuje i mała zmiana rozmiaru, powoduje dużą zmianę w szybkości wykonania algorytmu.

2.2 Dynamic Programming

Programowanie dynamiczne (w tym przypadku algorytm Held-Karp) polega na :

1. Obliczeniu długości ścieżki od każdego wierzchołka do wierzchołka końcowego.
2. Dla wszystkich kombinacji wierzchołków o rozmiarach od 2 do n:
   (a) korzystając z wcześniej zapamiętanych ścieżek o rozmiarze o jeden mniejszym, obliczyć długość nowej ścieżki,
   (b) ze ścieżek idących przez te same wierzchołki, zapamiętać tę najkrótszą.
3. Ostateczne ścieżki połączyć ponownie z wierzchołkiem końcowym, tak aby powstał cykl. Dodać długość tej krawędzi to długości ścieżki.
4. Z obliczonych ścieżek wybrać tę najkrótszą.

2.2.1 Algorytm i jego przebieg

W algorytmie najpierw zadeklarowano zmienną C jako słownik (1), którego kluczem będzie touple, który zapamiętuje binarnie zapisane odwiedzone wierzchołki oraz ostatni przyłączony do niego wierzchołek. Do takiego klucza przypisywany jest touple zawierający długość dotychczasowej ścieżki oraz wierzchołek, z którym połączono wierzchołek z klucza. Na początku połączono wszystkie wierzchołki z wierzchołkiem końcowym (w tym wypadku z 0) i zapisano je do słownika C (3-4). Następnie utworzono pętlę, która będzie zwiększać rozmiar ścieżki od 2 do n (6). Następnie utworzono pętlę, która będzie iterować po wszystkich kombinacjach o podanym rozmiarze (8). Następnie utworzono zmienną bits, która będzie zapamiętywała na poszczególnych pozycjach nr wierzchołków z danej kombinacji (9-12). Następnie utworzono pętlę, która dla każdego wierzchołka z danej iteracji (14):

- usunie go ze zmiennej bits (15), aby otrzymać kombinację o rozmiar mniejszą, którą już zapamiętaliśmy w słowniku C,
- stworzy pustą listę res (17), w której zapamiętamy długości nowych ścieżek,
- utworzy pętlę, z pozostałymi wierzchołkami z danej iteracji (19-22),

– doda dystans idący od wierzchołka iterowanego w pętli z linii 19, do wierzchołka z pętli z linii 14, do poprzedniego, już zapisanego wcześniej dystansu i wierzchołku końcowym z pętli z linii 19, do listy res (24),

- z listy res wybierze najkrótszą ścieżkę i doda ją do słownika C (26). Następnie do wszystkich ścieżek o rozmiarze n, połączymy ostatni wierzchołek z pierwszym, aby stworzyć oczekiwany cykl (31-32).

Z tych ścieżek wybrano najkrótszą, a na podstawie rodzica zapisanego przy danym kroku, odtworzono przebieg najkrótszej ścieżki, uzupełniając go o wierzchołek 0 (34-45).

Listing 4: Algorytm Dynamic Programming

```
1  C = {}
2
3  for k in range(1, n):
4      C[(1 << k, k)] = (arr[0][k], 0)
5
6  for subset_size in range(2, n):
7
8      for subset in itertools.combinations(range(1, n), subset_size):
9          bits = 0
10
11         for bit in subset:
12             bits |= 1 << bit
13         for k in subset:
14             prev = bits & ~(1 << k)
15             res = []
16
17             for m in subset:
18
19                 if m == k:
20                     continue
21
22                 res.append((C[(prev, m)][0] + arr[m][k], m))
23
24             C[(bits, k)] = min(res)
25
26 bits = (1<<n) - 2
27 res = []
28
29 for k in range(1, n):
30     res.append((C[(bits, k)][0] + arr[k][0], k))
31
32 opt, parent = min(res)
33
34 path = []
35
36 for i in range(n - 1):
37     path.append(parent)
38     new_bits = bits & ~(1 << parent)
39     _, parent = C[(bits, parent)]
40     bits = new_bits
```

```
41
42 path.append(0)
43 path.insert(0,0)
```

Złożoność algorytmu Dynamic Programming to $O(n^2 2^n)$, co czyni go algorytmem wydajniejszym od algorytmu Brute Force.

2.3 Branch and Bound

Algorytm Branch and Bound (inaczej metoda podziału i ograniczeń) to metoda polegająca na stopniowym kontynuowaniu ścieżki z wierzchołka o najmniejszym koszcie, które jednocześnie staje się dolnym ograniczeniem, stopniowo zwiększając dolne ograniczenie oraz na odcinaniu ścieżek, których koszt w pewnym momencie stanie się większy, niż górne ograniczenie.

W zastosowanym algorytmie, dolne ograniczenie jest obliczanie poprzez redukcję macierzy uprzednio przygotowanej, tj. takiej, w której niemożliwe już gałęzie, w danej odnodze, są usuwane. Poniższa funkcja redukująca macierz dostaje jako argument macierz, w której pozbyto się już zbędnych ścieżek. Funkcja ta każdą kolumnę i każdy wiersz redukuje o minimalną wartość w tej kolumnie lub w tym wierszu. Suma zredukowanych wartości stanowi dodatkowy koszt.

Listing 5: Redukcja Macierzy

```
1  def ReduceMatrix(array):
2      reduceValue = 0.0
3      for i in range(array.shape[0]):
4          minValue = np.nanmin(array[i])
5          minValue = np.nan_to_num(minValue)
6          reduceValue += minValue
7          array[i] -= minValue
8      for i in range(array.shape[1]):
9          minValue = np.nanmin(array[:,i])
10         minValue = np.nan_to_num(minValue)
11         reduceValue += minValue
12         array[:,i] -= minValue
13     return reduceValue, array
```

2.3.1 Algorytm i jego przebieg

Na początku algorytmu (1-13) zapamiętujemy w liście *all_data* toupla, który przechowuje pierwszy koszt redukcji, ilość jeszcze niezdobytych wierzchołków, bitowo zapisane zdobyte wierzchołki, unikalny indeks, pierwszą zredukowaną macierz, listę niezdobytych jeszcze wierzchołków, numer bieżącego wierzchołka oraz dotychczasową ścieżkę. Dane zapisane są w tej kolejności, ponieważ wyszukując najlepszego wierzchołka do rozwinięcia, należy wybrać najpierw ten z najmniejszym kosztem, następnie ten z największą ilością już dołączonych wierzchołków. Następnie zapisujemy liczbę, w której wszystkie wierzchołki są już zapisane (16). Jest to konieczne, aby sprawdzić, czy w pobranej ścieżce o najmniejszym koszcie, występują już wszystkie wierzchołki.
W tym momencie zaczyna się główna pętla algorytmu, która trwa do momentu spełnienia warunku końcowego (21). Na początku pobierane są dane o najmniejszym koszcie z listy *all_data* i sprawdzane, czy to już koniec algorytmu. (19-22), dane te są zapisywane w zmiennych (24-29), a liczba wierzchołków do których jeszcze nie doszliśmy, jest zmniejszana o 1 (31). W tym momencie, dla wszystkich wierzchołków, które wychodzą z danego węzła (33) obliczane są:

- macierz redukcji oraz koszt (35-40),

- bitowo dodawany jest bieżący wierzchołek (42-43),
- z wierzchołków nieodwiedzonych usuwany jest obecny wierzchołek (45-46),
- ścieżka rozwijana jest o bieżący wierzchołek (48-49),
- przydzielany jest nowy indeks (51),

oraz wszystkie te dane zapisywane są jako nowy touple w liście *all_data*.
Po przejściu przez pętlę usuwany jest dany węzeł wraz z danymi, ponieważ, do niego już nie będzie się wracało.
Po zakończeniu głównej pętli, naszym wynikiem jest minimalny touple z listy *all_data*

Listing 6: Algorytm Branch and Bound

```
1  first_reduce_cost, first_reduce_arr = ReduceMatrix(arr.copy())
2  first_bits = 1
3  vertices = np.arange(1,n)
4  vertices = list(vertices)
5  number_of_vertices = n-1
6  my_vertice_number = 0
7
8  index = 0
9
10 path = list()
11 path.append(0)
12 all_data = list()
13 all_data.append((first_reduce_cost, number_of_vertices, first_bits, index, first_reduce_
14
15
16 end_value = (1<<n) - 1
17
18 while True:
19     data = min(all_data)
20     old_bits = data[2]
21     if old_bits == end_value:
22         break
23
24     old_reduce_cost = data[0]
25     old_number_of_vertices = data[1]
26     old_reduce_arr = data[4]
27     old_vertices = data[5]
28     parent = data[6]
29     old_path = data[7]
30
31     number_of_vertices = old_number_of_vertices - 1
32
33     for vertice in old_vertices:
34         new_arr = old_reduce_arr.copy()
35         new_arr[parent] = np.nan
36         new_arr[:,vertice] = np.nan
37         new_arr[vertice,parent] = np.nan
38         reduce_cost, reduce_arr = ReduceMatrix(new_arr.copy())
39         reduce_cost = old_reduce_arr[parent, vertice] + old_reduce_cost + reduce_cost
```

```
40
41              bits = old_bits
42              bits |= 1 << vertice
43
44              vertices = old_vertices.copy()
45              vertices.remove(vertice)
46
47              path = old_path.copy()
48              path.append(vertice)
49
50              index += 1
51
52              all_data.append((reduce_cost, number_of_vertices, bits, index, reduce_arr, verti
53
54      all_data.remove(data)
55
56  data = min(all_data)
57
58  path = data[-1].copy()
59  path.append(0)
```

W algorytmie nie użyto górnego ograniczenia, które mogło by pozwolić zmniejszyć ilość danych zapamiętywanych w liście *all_data*.

Złożoność obliczeniowa algorytmu nie jest jednoznaczna, ponieważ zależy od użytego w algorytmie sposobu obliczania dolnego oraz górnego ograniczenia. Dodatkowo szybkość obliczenia dane przykładu, w dużej mierze zależy od jego danych.

3 Eksperymenty obliczeniowe

Obliczenia zastały wykonane na laptopie z procesorem i5-7300HQ, kartą graficzną NVIDIA GeForce GTX 1050, 16GB RAM i DYSK SSD. Algorytmy zostały napisane w języku Python, co wpłynęło na rząd wielkości uzyskanych wyników. Jako miarę jakości algorytmu przyjęto średnie procentowe odchylenie (Percentage Relative Deviation, PRD) najlepszego otrzymanego rozwiązania π względem rozwiązania referencyjnego π^{ref}:

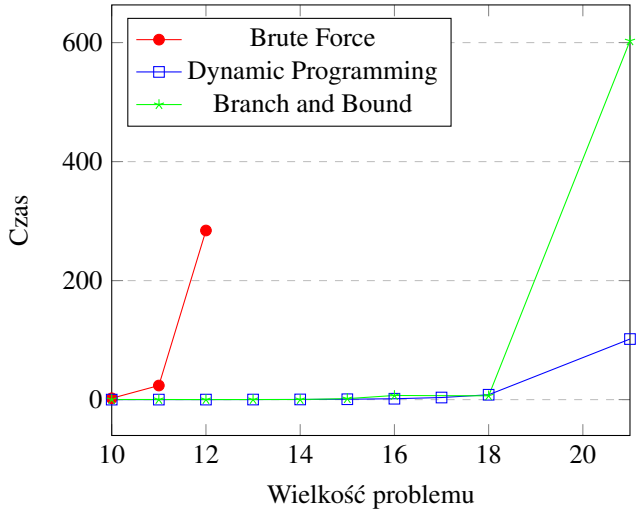
$$PRD(\pi) = 100\%(C_{max}(\pi) - C_{max}(\pi^{ref}))/C_{max}(\pi^{ref}) \tag{1}$$

Wszystkie wyniki zebrano i przedstawiono w tabeli nr 1 gdzie:

- n - liczba wierzchołków w grafie,
- $PRD_{bf}(\%)$ - średnie procentowe odchylenie dla algorytmu Brute Force
- $PRD_{dp}(\%)$ - średnie procentowe odchylenie dla algorytmu Dynamic Programming
- $PRD_{bb}(\%)$ - średnie procentowe odchylenie dla algorytmu Branch and Bound
- $t_{bf}(s)$ - czas dla algorytmu Brute Force,
- $t_{dp}(s)$ - czas dla algorytmu Dynamic Programming,
- $t_{bb}(s)$ - czas dla algorytmu Branch and Bound,

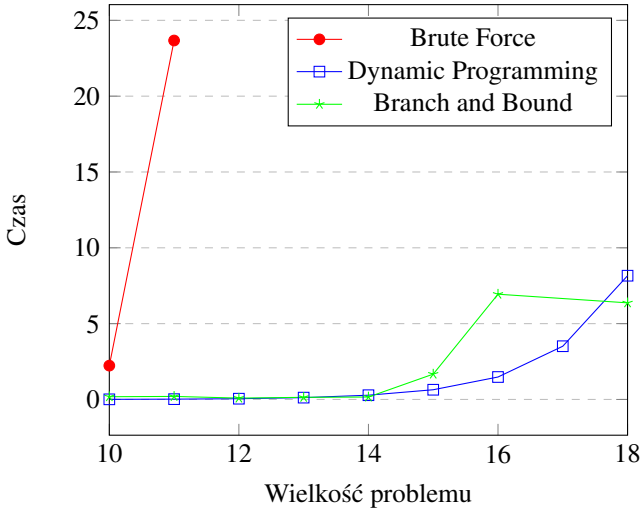
Tablica 1: Czas obliczeń oraz PRD dla ustalonej liczby iteracji.

n	$PRD_{bf}(\%)$	$PRD_{dp}(\%)$	$PRD_{bb}(\%)$	$t_{bf}(s)$	$t_{dp}(s)$	$t_{bb}(s)$
10	3.58	15.37	5.59	2.23	0.0082	0.18
11	2.26	13.57	4.93	23.67	0.020	0.20
12	2.71	12.88	4.77	284.21	0.049	0.089
13		10.65	5.47		0.12	0.14
14		10.38	5.25		0.28	0.16
15		8.85	2.05		0.64	1.67
16		7.83	1.25		1.48	6.94
17		9.64			3.51	
18		9.80	1.14		8.16	6.37
21		1.88	0.62		101.80	603



Rysunek 1: Czas wykonywania się algorytmów w zależności od wielkości problemu

Jak widać, niektóre punkty pomiarowe powodują, że wykres staje się nieczytelny, dlatego poniżej znajduje się wykres pozbawiony tych pomiarów.



Rysunek 2: Czas wykonywania się algorytmów w zależności od wielkości problemu

Algorytm Brute Force osiągnął wysokie czasy już dla początkowych wielkości problemu. Jest to bardzo prosty algorytm, ale dla wielkości problemu powyżej 10, nie daj już zadowalającego czasu.

Algorytm Dynamic Programming działa zdecydowanie lepiej niż algorytm Brute Force, jednak jego funkcjonalność również ma swoją granicę, którą jest ok dwukrotnie większa wielkość problemu niż dla algorytmu Brute Force.

Algorytm Branch and Bound okazał się algorytmem zaskakującym. Z losowo wygenerowanymi grafami, radził sobie podobnie co algorytm Dynamic Programming, jednak jeżeli wagi w grafie okażą się dla niego złośliwe, wtedy algorytm wykona obliczenia w czasie podobnym do algorytmu Brute Force.

4 Wnioski

Algorytmy dokładne dostarczają optymalne rozwiązania, jednak jak widać po pomiarach, bardzo szybko zwiększają czas przetwarzania algorytmu. Ich wykorzystanie ma jedynie sens, kiedy szukamy rozwiązania problemu o niewielkiej wielkości. Dla problemów o dużej wielkości, możemy nie doczekać się wyników. Język Python to dobre narzędzie do przeprowadzania pomiarów. Jego prostota pozwala skupić się na funkcjonowaniu algorytmu, jego ścieżkach rozwoju, bądź innych alternatywnych algorytmach, a nie na szczegółach wykonania kodu. Pozwala w łatwy sposób grupować dane i tworzyć z nich zestawienia. Jednak z powodu gorszych wyników, niż w przypadku napisania algorytmu w języku C++, algorytmy napisane w języku Python nie nadadzą się do zastosowań, w których każda sekunda się liczy.

Literatura

[1] Branch and bound. https://www.youtube.com/watch?v=1FEP_sNb62k. Accessed: 2019-11-21.

[2] Dynamic programming. https://www.math.uwaterloo.ca/ bico/papers/comp_chapterDP.pdf. Accessed: 2019-11-14.

[3] Eduinf. https://eduinf.waw.pl/inf/alg/001_search/0140.php. Accessed: 2019-11-07.

[4] Heap permutation. https://www.geeksforgeeks.org/heaps-algorithm-for-generating-permutations/. Accessed: 2019-11-07.